Providing secure access through network firewalls

[30990128]

Technical Field

This invention relates to methods and apparatus for providing secure access between a service external to a network firewall (such as a web page server) and a client internal to the firewall (such as a web browser).

Background Art

Facilities for global access to information, such as the World Wide Web, are undergoing immense growth and expansion, both in volume of data transferred and in sophistication of tools, such as web browsers, for accessing those data. This in turn requires the flexibility and capability of the data network infrastructure to be increased, without at the same time jeopardising its security features. Often these objectives conflict.

Web based applications are "client-server" in nature. A client-based browser runs on an end user's computer, communicating via network links with a web server operated by a service provider. The server provides data defining the content of web pages which are rendered and displayed by the user's browser; typically the web page content is defined using a markup language such as Hypertext Markup Language (HTML). The communications between the browser and the web server are conducted in accordance with a protocol called Hypertext Transfer Protocol (HTTP), defined in the Internet Engineering Task Force's RFC 1945. HTTP is a simple text-based protocol that has the peculiar quality that messages conforming to it are trusted and allowed to pass around the internets, intranets and extranets that make up today's "Internet". This Internet is really a series of closely coupled networks linked together through "firewalls": network nodes that allow controlled, restricted access between two networks. However the ability to "browse the world wide web" is seen as a universal common denominator, and as such HTTP messages are allowed to pass through these firewalls unchecked. There have been a number of enhancements to the HTTP protocol: the description herein relates by way of example to the basic version, HTTP/1.0, which is supported universally and with which later versions of HTTP are backwards compatible. Nonetheless the invention is not limited to use with HTTP/1.0 or indeed any other specific version of HTTP, and the claims hereof should be construed accordingly.

Web pages defined using the markup language can be enhanced by the use of code written in the Java programming language; this allows dynamic content and interactivity to be added to web pages. Java components can run either on the server end of the network connection, as "servlets", or on the client browser machine, in which case they are known as "applets". Many potential security problems were envisaged with the introduction of applets, such as "viruses" and "trojan horses", so a tight set of security restrictions were imposed on what applets could and could not do. To this end applets are executed on the client machine

in a controlled environment called a "sandbox". This sandbox defines how the applet can interact with the resources available in the computational platform the applet is running on, via a limited application programming interface (API). For example, the applet typically cannot interact with the local disk, nor connect to other computers on the network in unrestricted fashion. However the applet can typically connect back to the web server it was served from, although by way of an HTTP connection only. References to Java herein relate by way of example to Java/1.1 which is widely deployed; later versions of Java provide enhanced network support, but are backwards compatible with this base version.

The first generation of web content was mainly very static in nature – like pages from a magazine with text and pictures. The second generation of web content became increasingly dynamic, providing a user interface for applications, such as database queries. The third generation of web content is becoming increasingly interactive, with real-time communications, such as video, text chat and Internet Telephony, being added as an integral part. Internet-based client-server applications normally operate by opening "sockets" between the client and server, using Transaction Control Protocol/Internet Protocol (TCP/IP). TCP/IP sockets provide bi-directional, reliable communications paths. These can be used to implement dynamic or interactive client-server based applications, such as are required by the second and third generations of web content.

However, these more sophisticated applications pose a problem, in that the communications protocols they often require for interaction with the web server, such as TCP/IP, are intentionally barred by firewalls and proxy servers. It is therefore an object of this invention to provide clients, such as Java applets in a sandbox, with some controlled ability (within the context of a specific web-based application) to interact through a firewall with other resources, such as web servers, using protocols in addition to HTTP.

Disclosure of Invention

According to one aspect of this invention there is provided a method of permitting secure access between a service external to a network firewall and a client internal to the firewall, comprising the steps of:

- (a) effecting an HTTP GET operation or equivalent thereof (which may be in a protocol functionally equivalent to HTTP) from the client to establish a communications socket for communicating data from the service to the client;
- (b) after a predetermined interval effecting another GET operation or equivalent thereof to close the communications socket, irrespective of whether access between the service and the client is required to continue; and
- repeating steps (a) and (b) while access between the service and the client is required to continue.

The HTTP Tunneling Socket described herein as an example of this invention is a

socket that operates on top of the HTTP protocol. For interaction with a Java-based application it provides the standard APIs as defined for a socket in the Java/1.1 API specification. For interaction with the network it uses the HTTP/1.0 protocol. This technique in which one protocol is carried in messages conveyed over another protocol is known as tunneling. The use of tunneling has two benefits. It allows the tunnel to be executed in the browser sandbox, since the APIs for accessing an HTTP connection to a web server do not have any security restrictions. Secondly it allows the client-server connection to traverse firewalls between intranets and internets, as HTTP is the protocol that all firewalls allow to pass through them.

The Java applet and the server are thus provided with a connection protocol (such as TCP/IP) which is more versatile and capable than the HTTP connection to which they would normally be limited by the firewall and/or proxy server. However, the restrictions imposed by the sandbox prevent malicious or inappropriate access to the client machine's resources through the tunnelled protocol.

WO 99/50763 PCT/GB99/00900

Brief Description of Drawings

A method and apparatus in accordance with this invention, for permitting secure access between a client behind a firewall and a service provided by a node (such as a server) external to the firewall, will now be described, by way of example, with reference to the accompanying drawings, in which:

Figure 1	is a schematic illustration of circumstances in which the invention can be used;
Figure 2	shows an outline of an implementation of the invention in the Java programming language;
Figure 3	illustrates the basic principles of operation of the invention;
Figure 4	shows in more detail the mechanism for communications from the client to the server; and
Figure 5	shows in more detail the mechanism for communications from the server to the client.

Best Mode for Carrying Out the Invention, & Industrial Applicability

There are in principle two aspects involving firewalls to be considered when establishing a connection between an applet in a user's terminal and a web server. The first aspect is the case of a firewall-protected web server to which a browser running on a machine outside the firewall is trying to connect. This aspect does not present a serious problem since the web-server owner can configure the firewall to allow this connection to be made.

The second aspect is the case of a web browser machine in a corporate intranet trying to connect through the corporate firewall to a web server on the public Internet, as illustrated

in Figure 1. The problem here is that the web server owner has, by design, no control over the firewall. The complete scenario is that the browser 10 running java applet code on the user's terminal will connect initially to a machine 12 called a proxy within the corporate intranet 14. This proxy makes the required connections through the firewall 16 to the internet-based web server 18 on behalf of the browser 10, without exposing sensitive information about the browser and its host terminal to view in the public internet. The proxy 12 also performs various security checks, for example to ensure that the requests are valid HTTP requests.

However, because of the restriction to the use of HTTP, the web server is unable to provide desirable services to the client, such as real-time audio, video or interactive chat, which are possible with the use of more powerful protocols such as TCP/IP. These more powerful protocols are normally barred from unrestricted use because of the threat they pose of enabling inappropriate or intrusive access to resources within the corporate intranet.

This problem is overcome by using the invention to establish a "tunnel" through the firewall, carried by the HTTP messages, and thus open communications sockets which the Java applet and the web server can use to communicate with one another. In this way the web server is enabled to use resources on the browser's host machine, albeit still within the tight constraints imposed by the Java sandbox.

The tunnel consists of two Java programming classes, one (called the TunnelSocket) which implements the "Socket" class as defined in the Java/1.1 specification, and another (called TunnelServerSocket) which implements the "ServerSocket" class defined in that specification. As shown in Figure 2, client-based Java code can access the standard APIs (such as open, read, write, close, etc.) provided by a class conforming to the Socket class definition to communicate with a another Java application, such as one running on the web server. The server-based Java application can likewise use the same Socket APIs, plus the standard APIs such as accept etc. provided for servlets by the ServerSocket class, to communicate with the client-based Java applet. Messages between the Socket class and the ServerSocket class are carried within HTTP messages which traverse the firewall between the client and the server.

The HTTP tunnel uses the underlying simple text-based HTTP protocol to produce a reliable connection-based protocol between the TunnelSocket and TunnelServerSocket classes. Basic HTTP transactions use one of two methods in the HTTP request message. The GET method is designed to "get" a web page from a web server and the POST method is used to send data to a remote web resource or web application. In implementing the present invention, a GET based protocol is used for the client to server direction, and a POST based protocol is used for the reverse, server to client, direction, as shown in Figure 3. In the event that HTTP is supplanted by a functionally equivalent protocol, it is envisaged that the invention could be used with operations in that protocol equivalent to the HTTP GET and POST operations.

A single connection between the TunnelSocket and TunnelServerSocket classes is likely to comprise multiple HTTP transactions. To associate these transactions together, a numeric Globally Unique ID (GUID) is generated using a combination of random numbers, network address and time of day, so that each GUID generated is a unique quantity both in relation to time and location in the network. This GUID is incorporated in each HTTP request, both GET and POST, so that the HTTP protocol can recognize that they relate to the same communications socket. Specifically, the GUID is passed as the HTTP Uniform Resource Indicator (URI) (the field used to specify a document name in normal use of HTTP).

Client to server connection is implemented using the POST HTTP operation. Each time a write is performed to the client socket (TunnelSocket class) the request is packaged up and sent as the payload of an HTTP POST message, including the relevant GUID, as shown in Figure 4. Each such write operation sends a separate POST message through the firewall.

The server to client direction is more complex because of restrictions imposed by the proxy server 12 and the firewall 16. As shown in Figure 5, the client issues an HTTP GET message which tells the server that a new tunneled socket is being established. The HTTP GET request will generate a temporary server-client socket that allows data to be written from the server to the client. However as a security precaution most proxy server implementations will close down such a server-client socket after a time interval *P*, typically of the order of ten minutes in duration. Accordingly the tunnel established in this invention periodically itself forces a pre-emptive close down of the connection at some periodicity *T* which is shorter in duration than the interval *P*. This is accomplished by issuing another GET request, to force the TunnelServerSocket class to close the existing socket to the client and immediately establish a new socket. It should be noted that this closure (and subsequent re-opening) are performed even though data are still pending to be transferred from the server to the client. This approach avoids the overhead of previous techniques, in which the client polls the server at some preselected polling interval, usually of the order of 20 milliseconds. These frequent polls are equivalent to multiple web page requests, so the server receives a large number of web hits per second, imposing a significant overhead and severely restricting the scalability of such prior techniques to handle large numbers of clients simultaneously.

An illustration of the implementation of the invention is provided below, in the form of pseudo-code, which emphasizes the significant aspects of the implementation but for the sake of clarity omits minor details which, although required in practice, are well within the capability of a person skilled in the art. Likewise the existence is assumed of certain subsidiary functions, such as guidFactory for creating a GUID and createFiFoInputStream for reserving and configuring memory for use as a first-in-first-out (FIFO) buffer; again these functions are individually well known in the art, so their internal details do not need to be specified. Depending on the particular manner of implementation, additional parameters may

0605369 .060500

be included, for example, in some function calls; this is indicated by an ellipsis (...).

1. Server to Client direction.

1a. TunnelSocket InputStream Code – runs on client

```
//
// Open an Input Stream to the Server
// This is the Server to Client direction of the Tunnel Socket.
//
// serverAdresss - Internet address e.g. fred.hpl.hp.com:
//    standard socket
// Port number is a port number on the server machine to
//    connect to: standard socket
public   open(serverAddress, portNumber) {

   // Generate a new Globally unique id for the new socket.
   // The id is unique across the whole network and across
   //    time.
   // It is generated using a random number generator, the
   //    node's network address and the current time.
   guid = guidFactory();

   // Create a Fifo stream that is used by the application code
   //    to read data.
   // The underlying protocol reads data off the wire into this
   //    Fifo ready for the application.
   fifoInputStream = createFifoInputStream();

   // Issue an HTTP/1.0 GET request to the server
   // The on-the-wire format is:
   //    GET <guid>?command=establish
   //    Content-Type=application/octet-stream
   inputStream = issueHttpGetRequest(uri=guid,
      queryString="command=establish",
      Content-Type="application/octet-stream",
      address=serverAddress, port=port);

   // Start a timer thread that is invoked at a periodicity of
   //    GETTimeOut seconds.
   // Typically GETTimeOut is a few minutes in duration - but
   //    it must be less than the time out period of the proxy.
   // The function switchGETStream() is called at the
   //    periodicity with its own thread of execution.
   startPeriodicTimeOutDemon(switchGETStream(), GETTimeOut);

}


// Send a new request to the server telling it to close down
//    the current stream and establish a new stream
//    periodically
```

```
private  switchGETStream() {

            // Lock the Input Stream to prevent any race conditions with
            //    readInputStream
            lock(inputStreamLock)

            newInputStream = issueHttpGetRequest(uri=guid,
               queryString="command=switch",
               Content-Type="application/octet-stream",
               address=serverAddress, port=port);

            // Drain the contents of the stream until it is closed by
            //    the remote end
            while (read(InputStream, FifoStream) != EOF);
               writeToFifo(FifoInputStream);

            inputStream = newInputStream();
            unlock(inputStreamLock);

            return();

         }


         // Read data from the stream via the Fifo - the Fifo buffers
         //    data up ready to be read
public   readInputStream(..., buf, len) {
         ...
            // if there isn't enough data in the Fifo to satisfy this
            //    read request - top it up from the wire.
            if (!FifoStream.length() > len) {

               //Transfer data from the stream established to the Server
               lock(inputStreamLock);
               writeToFifo(FifoInputStream, read(InputStream,
                  len - FifoStream.length());
               unlock(inputStreamLock);
            }

            // return data in from Fifo
            return(read(FifoInputStream, ..., buf, len)));

         }


         //
public   close() {

            // Stop timer thread
            stopPeriodicTimeOutDemon(switchGETStream());
```

0050-9950-0950

```
        // Mark stream as closed
        markInputStreamClosed();

    }
```

5

05050592.050500

1b. TunnelServerSocket OutputStream – runs on the server

```
// Server side Protocol engine handling all requests the server receives

        // Sit waiting on a ServerSocket for socket connections to come
        //    in … just standard socket call
        while ((socket = accept() still open) {

           httpheader = readHttpHeader(socket);

           if ( httpHeader.type == "GET" &&
              httpHeader.command = establish) {

              // New socket request
              // Open an output Stream
              outputStream = socket.getOutputStream();

              // Add an entry to the database of opened Tunnel Sockets
              addEntryTunnelSocketHashDataBase(socket, outputStream,
                 httpHeader.guid);


           } else ( httpHeader.type == "GET" &&
              httpHeader.command == "switch") {

              // Lookup tunnelSocket
              tunnelSocket =
                 lookupEntryTunnelSocketDatabase(httpHeader.guid);

              // Request to switch stream generated by the TunnelSocket
              //    timer on the client
              // Close the current output stream
              // Take lock to avoid conflict with
              //    tunnelOutputStreamWrite
              lock(tunnelSocket.outputStreamLock);
              close(tunnelSocket.outputStream);

              // establish a new output stream on this new connection.
              tunnelSocket.outputStream = socket.getOutputStream();
              unlock(tunnelSocket.outputStreamLock)
           } else (httpheader.type == "POST" )
              // See pseudo-code fragment included below with Server-
              //    Client direction.
              doPOST();

           }

           tunnelOutputStreamWrite(buf, len) {
           ...
           // Take lock to avoid conflict with "switch" in
           //    tunnelServerAccept()
           lock(outputStreamLock)
```

```
        write(outputStream, buf, len);
        unlock(outputStreamLock)
    }
```

.....

5

0950950 "950950

2. Client-Server direction.

2a. Client side TunnelSocket OutputStream

```
public   write(buf, length, ...) {

            // Issue an HTTP/1.0 POST request to the server
            // The on the wire format is:
            //    POST <guid>
            //    Content-Type=application/octet-stream
            //    Content-Length=<length>

            inputStream = issueHttpPOSTRequest(uri=guid,
               Content-Length=length,
               Content-Type="application/octet-stream",
               address=serverAddress, port=port, payload=buf);

         }
```

2b. Server side TunnelServerSocket InputStream

```
public   readServerSideInputStream(buf, len, ...) {

            return(readFromFifo(FifoServerSideInputStream, buf, len,..);

         }
```

```
         ...
         // On receiving a POST request (see 1b above)
         // copy the payload into the Fifo buffer ready for the
         //    readServerSideInputStream to read it.
         doPOST() {
            writeToFifo(FifoServerSideInputStream,
               httpRequest.readPayLoad(), httpHeader.contentLength);
         }
         ....
```

0550-2-9-2-0-5-0-5-3-6-0-2